

Notion de module en C et fichier header .h

Pré-requis

Vous savez comment compiler un code source situé dans un fichier .c avec `gcc`.
Vous connaissez les options `–c` et `–o` du compilateur `gcc`.

1. Notion de module en C

Un programme peut être constitué de plusieurs millions de lignes de code et de fonctions. Sans atteindre cette complexité, dès lors qu'on a plus, par exemple, d'une dizaine de fonction ou d'un millier de lignes de code le code, il devient essentiel pour s'y retrouver d'organiser votre code en plusieurs *modules.*

En `C`, on appellera *module* un couple de fichier *source* (extension `.c`) et *en-tête* (« header », extension `.h`). Un module regroupe des fonctions et/ou variables et/ou types traitant une sous partie du problème global.

Le découpage d'un projet logiciel `C` en modules, tout comme le découpage en fonctions, est fait au cas par cas en phase d'analyse. On cherchera à ce que chaque module corresponde à un tout signifiant. Par exemple, un module pourrait être consacré à la gestion d'un type abstrait donné (type liste chainée et fonctions de manipulation des listes chainées par exemple), un autre aux entrées/sorties pour un type de fichier, un autre pourrait regrouper les fonctions implantant un algorithme donné, etc.

2. Role des fichiers .c et .h

On considère ici un module « `monModule` », donc deux fichiers `monModule.h` et `monModule.c`.

Dans le fichier source `monModule.c` du module, on écrira le *code source* du module : instanciation des variables globales éventuelles, corps de toutes les fonctions du module.

Le fichier *header* `monModule.h` du module contiendra lui *tout ce que le compilateur doit connaître pour lorsqu'on compile du code qui utilise les fonctionnalités du module* `monModule`*.* En d'autres termes, le fichier *header* contient *non pas* les fonctionnalités du module `monModule` (qui sont elles dans le code source du fichier .c), mais uniquement le « mode d'emploi » de ces fonctionnalités, mode d'emploi dont le compilateur doit être informé lorsqu'on utilise ces fonctionnalités par exemple dans le code source .c d'un *autre* module qui utilise `monModule`.

Le fichier `monModule.h` regroupera donc en particulier :

- les définitions des *types publics* du module `monModule`
- les *prototypes* (ou signatures) des fonctions publiques du module
- les déclarations d'existence des *variables globales publiques* instanciées par le module



Par « public » ont entend, ici, « qu'il faut pouvoir manipuler pour utiliser le module `monModule` depuis un *autre* module ».

Outre qu'il est essentiel pour le *compilateur*, le fichier *header* aura aussi un rôle de « mode d'emploi » du module pour le développeur, car il liste toutes les fonctionnalités du module, sans dire comment ces fonctionnalités sont en fait réalisés. C'est pourquoi on appelle aussi le fichier header `monModule.h` "l'interface" du module `monModule.`

3. Inclusion d'un fichier .h avec #include

Pour utiliser les fonctionnalités de `monModule` depuis un autre module `autreModule`, il faut *inclure* le fichier header de `monModule` dans `autreModule`. Cela se fait avec la primitive de précompilation `#include` :

```
// fichier autreModule.c

#include "monModule.h"
```

Lorsque le compilateur rencontre une inclusion `#include` d'un fichier il va, avant de lancer la compilation proprement dite, *remplacer* cette inclusion par le contenu du fichier inclus.

Ainsi, dans l'exemple précédent, lors de la compilation de `autreModule.c`, tout se passera comme si le contenu du fichier d'en-tête `monModule.h` était collé à la place du `#include`… En conséquence, quand le compilateur compilera le fichier `autreModule.c`, il aura connaissance des déclarations contenues dans `monModule.h`. Par exemple, il aura connaissance des prototypes des fonctions de `monModule` (pas de leur corps, qui est dans `monModule.c`) et sera donc a même de vérifier si les appels de ces fonctions sont syntaxiquement corrects. Il vérifiera donc typiquement que chaque appel de fonction est fait avec le bon nombre de paramètres, et des types corrects pour chaque paramètre, etc. De même, il aura connaissance des types déclarés dans `monModule.h`, de telle sorte qu'il devient possible, par exemple, de déclarer des variable de ces types dans `autreModule.c`.

Remarques

Inclusion du header dans le fichier source du module. Vous avez fortement intérêt à ce que le fichier source .c d'un module inclut son propre fichier header .h (monModule.c inclut monModule.h). Cela permet que le compilateur vérifie, lorsque le module sera compilé, que les deux correspondent bien.

Inclusion des *headers* des librairies standard. Avec les explications qui précèdent, vous devriez maintenant comprendre pourquoi, au début d'un fichier source, on inclut les headers des fonctions de la librairies standard qui sont utilisées par le module : ces headers contiennent les types et prototypes des fonctions en question. Notez que, pour les inclusions des librairies standard, on utilie « < » « > » au lieu de double guillemets – par exemple `#include <maths.h>` ou `#include <stdio.h>`.

Inclusion d'un *header* .h dans un autre *header* .h. Si (et seulement si en général) un fichier *header* `monModule.h` utilise des déclarations d'un autre fichier `autreModule.h`, typiquement si `monModule.h` déclare des fonctions dont certains paramètres sont d'un type déclaré dans `autreModule.h`, il faut inclure `autreModule.h` dans `monModule.h`.



Primitive #define d'un fichier .h

Un fichier *header* .h débute *toujours* par un couple de primitive de pré-compilation

```
#ifndef IDENTIFIANT_UNIQUE

#define IDENTIFIANT_UNIQUE
```

où `IDENTIFIANT_UNIQUE` est un identifiant choisi arbitrairement mais qui doit être unique et #ifndef signifie « if not defined ».

```
#endif
```

Cela permet au compilateur de s'assurer que le contenu du fichier .h ne sera inclus qu'une seule et unique fois lors de la compilation.

En général, on choisit pour `IDENTIFIANT_UNIQUE` quelque chose qui évoque le nom du module. Par exemple, pour le module `monModule`, on pourrait choisir `_MON_MODULE_H_`.

4. Exemple de fichier header .h et .c

Exemple de fichier header .h pour monModule

```
// fichier monModule.h
#ifndef _MON_MODULE_H
#define _MON_MODULE_H

// pour connaitre le type LISTE_T déclaré dans liste.h
// et utilisé dans les prototypes des fonctions de monModule.h
#include "liste.h"

/* declaration d'existence de variables globales
 * publiques de monModule
 */

// Notez que ce n'est pas une instantiation,
// mais, du fait de l'utilisation du mot clé "extern"
// juste une déclaration d'existence de la variable.
// L'instanciation est, elle, faite dans le .c du module
extern int uneVariableGlobale;
extern LISTE_T uneVariableListeGlobale;

/* declaration des types publics de monModule
 */
typedef struct complex {
   float re;
   float im;
} COMPLEXE_T;

/* prototype des fonctions publiques de monModule
 */
void trier_liste(LISTE_T liste);
void conjuguer_complexe(COMPLEXE_T * p_unComplexe);

#endif
```



Exemple de fichier source .c pour monModule

```
/* fichier monModule.c */

//inclusion des librairies standards
#include <stdio.h>

// inclusion du header de monModule
// pour connaitre les types qui y sont déclarés
// et verifier les prototypes
#include "liste.h"

// inclusion des autres headers du projet dont on a besoin
#include "unAutreModule.h"

// instanciation et initialisation des variables globales
int     uneVariableGlobale=0;
LISTE_T uneVariableListeGlobale=NULL;

/* definition des fonctions */
void trier_liste(LISTE_T liste) {
   // corps de la fonction
   // ...
}

void conjuguer_complexe(COMPLEXE_T * p_unComplexe){
   // corps de la fonction
   // ...
}
```

5. Quelques cas particuliers de fichier *header*

Parfois, on pourra écrire un fichier .h sans ficher .c correspondant.

Cela est par exemple très utile lorsque on veut déclarer un ou des types qui seront ensuite utilisés dans de nombreux fichiers sources. On pourra alors créer un fichier `types.h` dans lequel on fera figurer la déclaration des types :

```
// fichier types.h
#ifndef _TYPES_H_
#define _TYPES_H_
// Déclaration des principaux types du projet
typedef double MesDouble_t ;
typedef struct { float x, y } MesComplexes_t
#endif
```

Ce fichier sera ensuite inclut dans tous les fichiers header .h des modules qui utilisent ces types.